

Functional Programming in Java (stream)

Processing Data with Java SE 8 Streams – Part1 & Part 2

<https://www.oracle.com/technetwork/articles/java/ma14-java-se-8-streams-2177646.html>

<https://www.oracle.com/technetwork/articles/java/architect-streams-pt2-2227132.html>

Stream 왜 사용하나?

- Collection 데이터 처리를 선언적(declarative)으로
 - SQL 형태의 연산 : finding, grouping
 - "SELECT id, MAX(value) from transactions"
 - No implementation of how to calculate (no loop using index)
 - Only expression what we expect.
- 처리 성능의 문제
 - 대량의 데이터를 빠르게 처리하자
 - Multicore를 이용한 병렬 프로그래밍

프로그래밍 비교 (예전 vs java8의 함수형)

```
List<Transaction> groceryTransactions = new ArrayList<>();
for(Transaction t: transactions){
    if(t.getType() == Transaction.GROCERY)
        groceryTransactions.add(t);
}

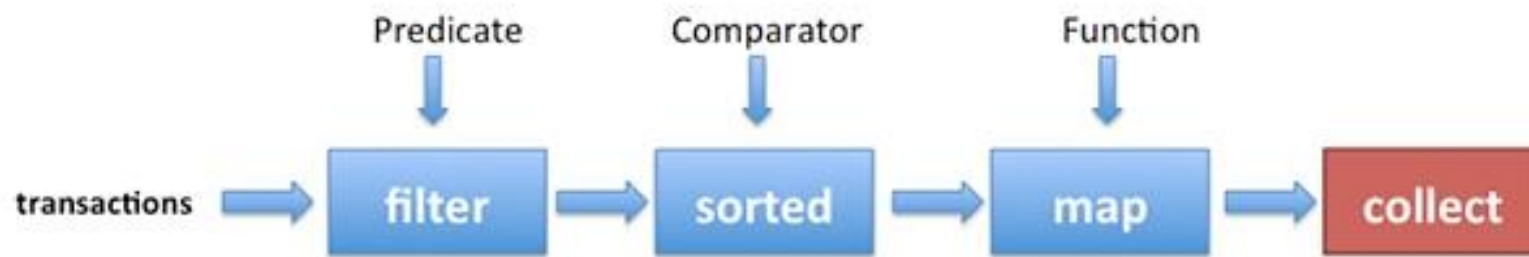
Collections.sort(groceryTransactions, new Comparator() {
    public int compare(Transaction t1, Transaction t2) {
        return t2.getValue().compareTo(t1.getValue());
    }
});

List<Integer> transactionIds = new ArrayList<>();
for(Transaction t: groceryTransactions){
    transactionIds.add(t.getId());
}
```

```
List<Integer> transactionsIds = transactions.stream()
    .filter(t -> t.getType() == transaction.GROCERY)
    .sorted(comparing(Transaction::getValue).reversed())
    .map(Transaction::getId)
    .collect(toList());
```

- 람다식(lambda expression)과 method reference 이용

Stream의 파이프라인 방식 처리



1. stream() 로 Collection을 스트림으로 변환
2. 스트림에서 동작하는 함수를 순차적(파이프라인)으로 적용
3. Collect에 의해 스트림의 데이터를 변환
 - toList() 를 적용하여 List로 변환

손쉬운 병렬 프로그래밍

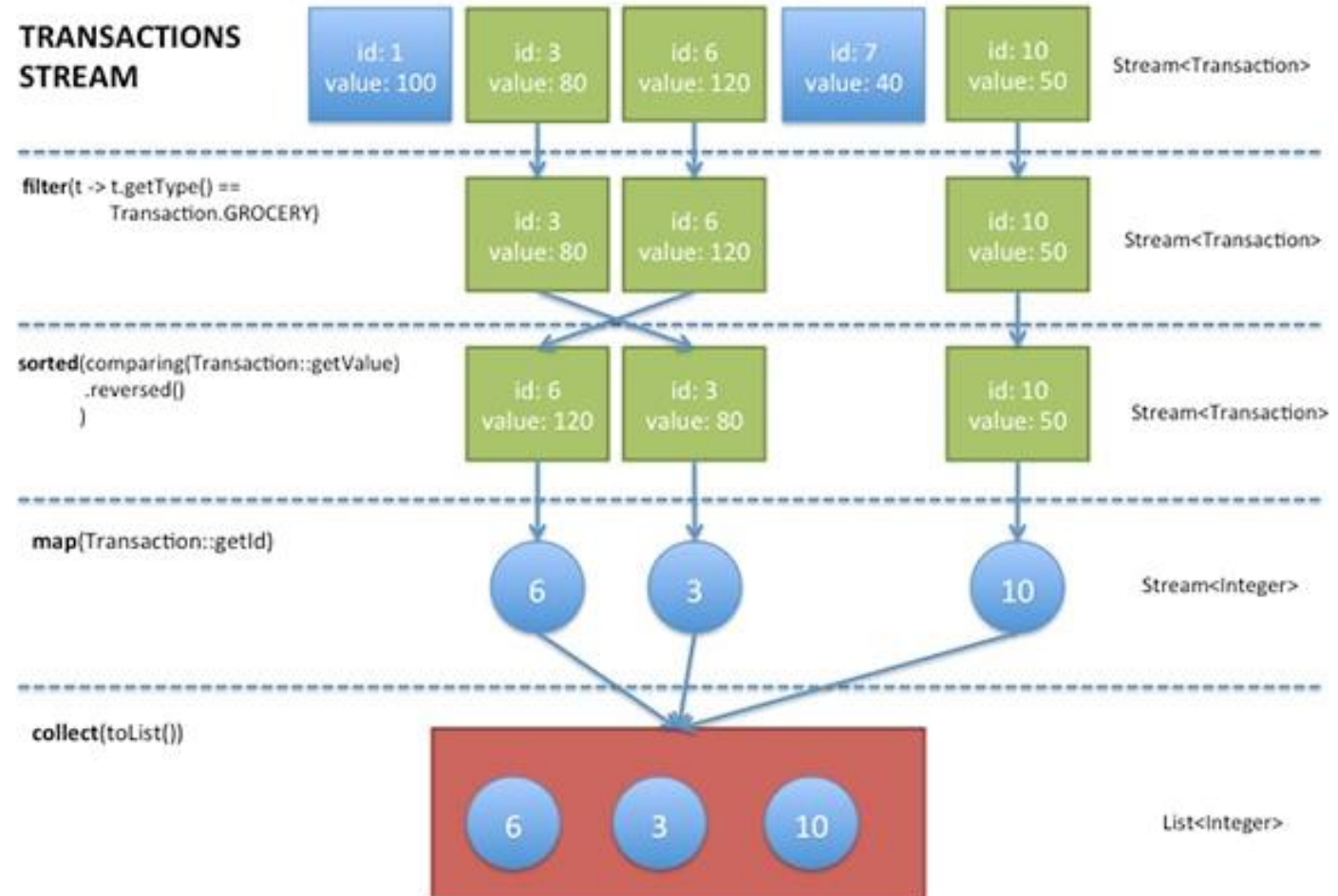
```
List<Integer> transactionsIds = transactions.stream()  
    .filter(t -> t.getType() == transaction.GROCERY)  
    .sorted(comparing(Transaction::getValue).reversed())  
    .map(Transaction::getId)  
    .collect(toList());
```

```
List<Integer> transactionsIds = transactions  
    .parallelStream()  
    .filter(t -> t.getType() == transaction.GROCERY)  
    .sorted(comparing(Transaction::getValue).reversed())  
    .map(Transaction::getId)  
    .collect(toList());
```

Stream이란?

- "a **sequence** of elements from a **source** that supports **aggregate operations**"
- 원재료(Source)
 - 데이터가 **저장**된 자원 : Collections, arrays, 파일 등의 입출력 자원들
- 순차적인 원소들(Sequence of elements)
 - 스트림은 데이터를 저장하지 않음
 - 저장된 각 데이터 타입의 원소들을 **순차적**으로 처리할 수 있는 Interface
- 데이터 뭉치(aggregate)를 처리하는 함수들
 - 순수 함수형 프로그램의 원리를 만족하는 함수
 - filter, map, reduce, find, match, sorted 등
- 스트림 오퍼레이션의 특징
 - 파이프라이닝 : 여러 함수들이 연속적으로, 순차적으로 적용됨
 - 내재된 iteration : 스트림 자체에 iteration 기능이 내재되어 있으므로, iteration을 **명시적**으로 표현할 필요가 없음 (collection의 경우와 비교됨)

```
List<Integer> transactionsIds = transactions.stream()
    .filter(t -> t.getType() == transaction.GROCERY)
    .sorted(comparing(Transaction::getValue).reversed())
    .map(Transaction::getId)
    .collect(toList());
```



Stream vs Collections

- 둘 모두 원소들의 순차적 접근을 위한 인터페이스 제공 (iterator)
- 차이점 : collection은 데이터에, 스트림은 계산에 중점을 둠
- 예 : DVD에 저장된 영화
 - 이 저장된 내용은 collection이다.
 - 이 내용이 인터넷 다운로드 등을 통해서 스트림으로 전달되는 상황은 stream이다.
 - 스트림에 전달되는 내용은 전체가 아닌, 앞 부분의 일부분 만을 계산하더라도 동작할 수 있다.
- 계산이 언제(when) 적용되나?
 - Collection : 한 원소를 collection에 첨가하기 위해서는 collection의 모든 원소들이 모두 계산이 완료된 후에 가능하다. (eager evaluation)
 - Stream : 필요에 따라 계산이 적용될 수 있다 (lazy evaluation)
- External vs Internal Iteration
 - Collection에서는 iteration이 for 혹은 forEach 등에 의해서 **명시적으로** 표현됨
 - Stream 에서는 내부적으로 iteration 및 계산 결과의 저장이 이루어짐

명시적 vs 묵시적 iteration

```
List<String> transactionIds = new ArrayList<>();  
for(Transaction t: transactions){  
    transactionIds.add(t.getId());  
}
```

```
List<Integer> transactionIds =  
    transactions.stream()  
        .map(Transaction::getId)  
        .collect(toList());
```

- Map은 주어진 함수를 스트림의 각 원소에 적용함
- Collect는 stream을 List로 변환함

Stream과 관련된 함수들(java.util.stream.Stream)

- 여러 함수들 중에서
 - filter, sorted 등 : 여러 함수들이 파이프라인 형태로 동작함. 각 함수의 수행 결과의 타입은 Stream 이며, 이는 전체 계산의 중간 결과임.
 - collect : 파이프라인을 종료하고, 계산 결과(전체 계산의 결과)를 복귀함. 복귀 값의 타입은 List, Integer, void 등이 될 수 있음. 이 오퍼레이션은 터미널 오퍼레이션 중의 하나로서, stream의 계산은 반드시 터미널 오퍼레이션으로 종료되어야 함.
타입 : boolean(allMatch 등), void(forEach), Optional(findAny 등)
- Lazy Evaluation
 - 중간 단계의 계산은 최종단계의 계산이 invoke되기 전까지는 수행되지 않는다.

1. 데이터 소스(collect 등)를 스트림으로 변환
2. 여러 중간 단계의 함수들을 적용함
3. 한 터미널 오퍼레이션으로 결과를 생성하고 종료함.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);
List<Integer> twoEvenSquares =
    numbers.stream()
        .filter(n -> {
            System.out.println("filtering " + n);
            return n % 2 == 0;
        })
        .map(n -> {
            System.out.println("mapping " + n);
            return n * n;
        })
        .limit(2)
        .collect(toList());
```

출력

```
filtering 1
filtering 2
mapping 2
filtering 3
filtering 4
mapping 4
```

Stream의 주요 함수

- Filtering : filter(Predicate), distinct, limit(n), skip(n)
- Matching (결과 값은 Predicate) : anyMatch, allMatch, noneMatch


```
boolean expensive = transactions.stream().allMatch(t -> t.getValue() > 100);
```
- Retrieving elements : findFirst, findAny (Optional : 원소의 존재여부 판단)


```
Optional<Transaction> = transactions.stream()
                                   .filter(t -> t.getType() == Transaction.GROCERY)
                                   .findAny();

transactions.stream()
    .filter(t -> t.getType() == Transaction.GROCERY)
    .findAny()
    .ifPresent(System.out::println);
```
- Mapping : map (입력으로 주어진 함수를 원소에 적용함)

- 누적 계산 : reduce (계산을 누적하면서 반복적으로 적용함)

```
int sum = 0; for (int x : numbers) sum += x;  
int sum = numbers.stream().reduce(0, (a, b) -> a + b); // Integer operation  
int product = numbers.stream().reduce(1, (a, b) -> a * b);  
int maximum = numbers.stream().reduce(1, Integer::max);
```

- Primitive operations - int, double, long

- mapToInt, mapToDouble, mapToLong

```
int statementSum =  
    transactions.stream()  
        .mapToInt(Transaction::getValue) // error if it were map  
        .sum(); // work on primitive int
```

- 수로 이루어진 스트림의 타입 : IntStream, DoubleStream, LongStream

- 스트림 숫자 생성 메소드 : range, rangeClosed

- 예 : IntStream oddNumbers = IntStream.rangeClosed(10, 30).filter(n -> n % 2 == 1);

- 배열로부터 스트림 생성

- Stream.of Arrays.stream

- 예

```
Stream<Integer> numbersFromValues = Stream.of(1, 2, 3, 4);
```

```
int[] numbers = {1, 2, 3, 4};
```

```
IntStream numbersFromArray = Arrays.stream(numbers);
```

- 파일로부터 스트림 생성 :

- Files.lines

- 예

```
long numberOfLines =
```

```
Files.lines(Paths.get("yourFile.txt"),Charset.defaultCharset()).count();
```

- 무한(infinite) 스트림 생성

- Stream.iterate Stream.generate

- 예

```
Stream<Integer> numbers = Stream.iterate(0, n -> n + 10); // 10, 20, ...
```

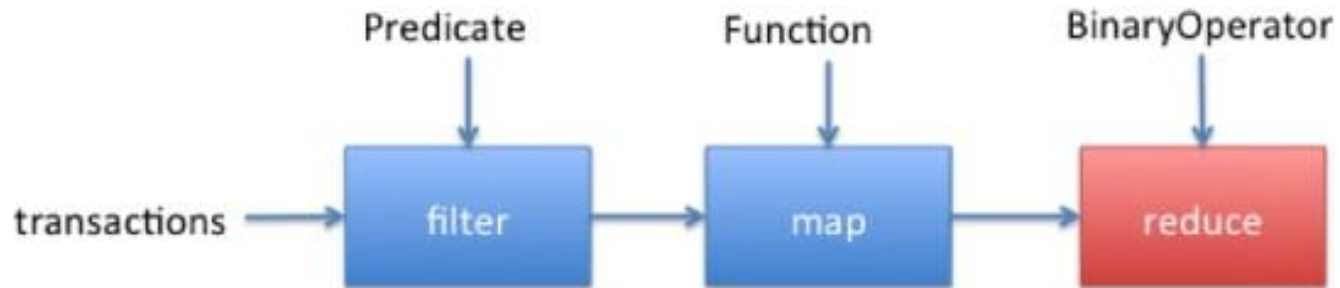
```
numbers.limit(5).forEach(System.out::println); // 0, 10, 20, 30, 40
```

고급 기능

Part 2: Processing Data with Java SE 8 Streams

- 예

```
int sumExpensive = transactions.stream()  
    .filter(t -> t.getValue() > 1000)  
    .map(Transaction::getValue)  
    .reduce(0, Integer::sum);
```



flatMap (flat . map)과 collect

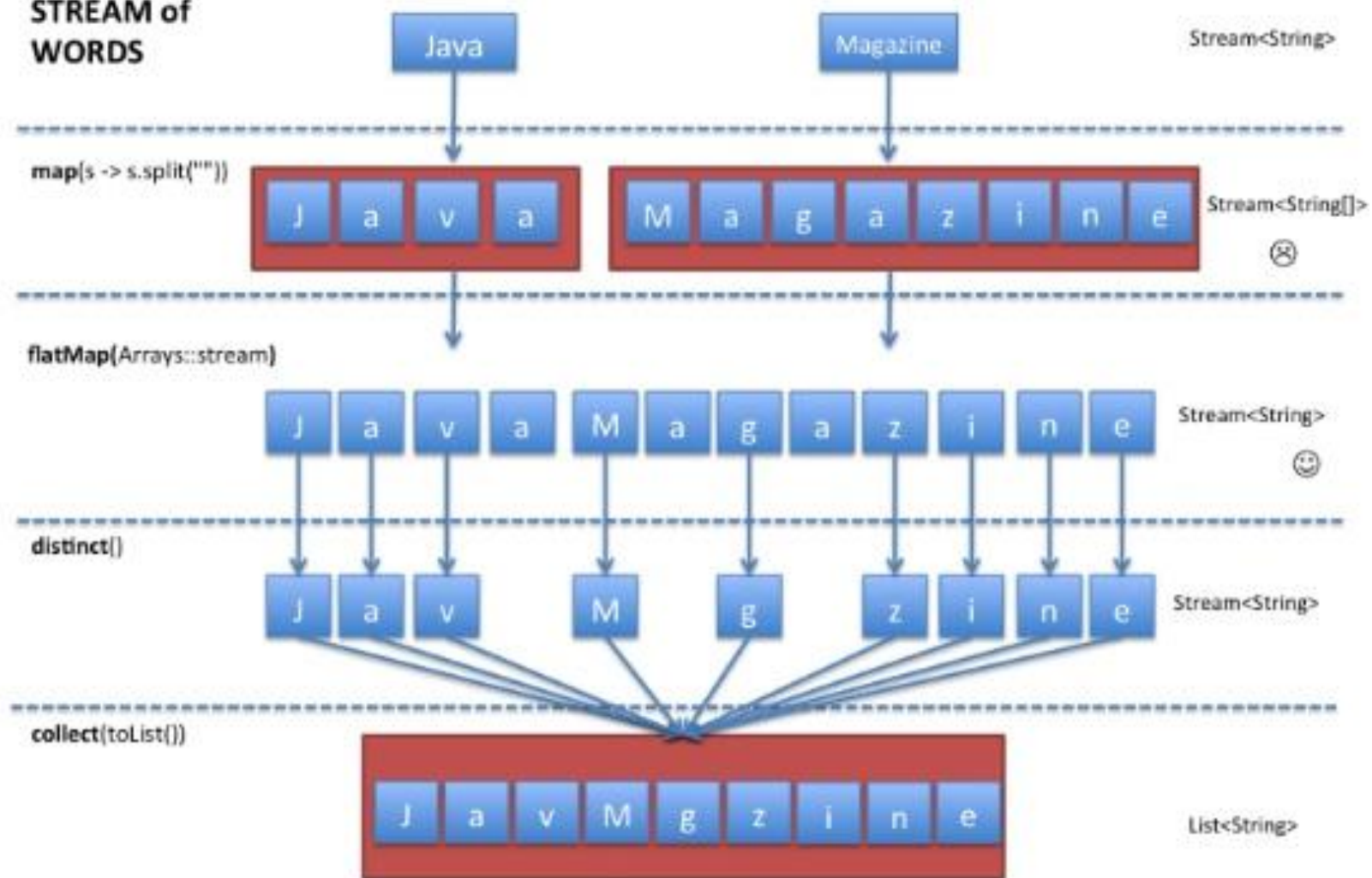
스트림 전체에 있는 문자의 수 카운트하기

```
import static java.util.function.Function.identity;
import static java.util.stream.Collectors.*;

Stream<String> words = Stream.of("Java", "Magazine", "is", "the", "best");
Map<String, Long> letterToCount =
    words.map(w -> w.split(""))
          .flatMap(Arrays::stream)
          .collect(groupingBy(identity(), counting()));
```

```
[a:4, b:1, e:3, g:1, h:1, i:2, ... ]
```


STREAM of WORDS



```
import static java.util.function.Function.identity;
import static java.util.stream.Collectors.*;

Stream<String> words = Stream.of("Java", "Magazine", "is", "the", "best");
List<String> newWords = words.map(w -> w.split(" "))           // Stream<String[]>
                             .flatMap(Arrays::stream)          // Stream<String>
                             .distinct()                        // Stream<String>
                             .collect(toList());                // List<String>
```

기본 개념, Haskell과의 비교

스트림 사용의 제약(consumed only once)

Java의 List(ArrayList/Linked List)와 Stream

- Java의 List
 - Mutable data structure
 - 순수 함수형 프로그래밍 기법을 적용하기 어려움
- 어떻게 List에 순수 함수형 프로그래밍을 적용할 수 있을까?
 - List를 stream으로 변환하고
 - Stream에서 순수 함수형 프로그래밍 기법을 적용하며
 - 필요에 따라 계산 결과를 List 형태로 변환함
 - 변환 함수
 - stream, of 등 : 리스트 -> stream
 - Collect의 toList : stream -> 리스트

Collection 형태의 데이터 처리

- 반복문
 - 배열, 리스트 등 여러 개의 데이터들로 구성된 collection 형태의 데이터를 처리하기 위해서는 loop 문 형태의 프로그래밍이 필수적임
 - 할당문을 사용하지 않는 순수 함수형 프로그래밍에서는 반복문 대신 재귀함수를 사용함
- Iterator
 - 다량의 데이터 처리를 위해 원소들을 하나씩 순서적으로 제공하는 기능이 필요함
 - Loop 문과 배열을 사용하는 경우 $a[i++]$ 형태의 index 및 index 값의 변화를 통해서 처리하고 있음
- 재귀 함수
 - 재귀 함수의 경우 iteration은 호출하는 파라미터의 값을 조정하거나 리스트 size를 변화시켜 iterator에 해당하는 기능을 함
- List Comprehension의 경우 -> 원소 generator가 iterator의 기능을 함

List Comprehension의 Iterator 기능

- Generator

- 한 (무한) 리스트의 원소를 맨 앞 (head) 부터 하나씩 순서대로 제공함 (즉 iterator 의 기능을 갖고 있음)
- 따라서 리스트의 한 원소의 접근은 **index를 사용하지 않으며**, 예를 들어, 10번째 원소를 얻으려고 할 때는 앞 부분의 9개 원소들을 모두 방출한 후 10번째 원소를 접근할 수 있음
- 한 리스트의 원소 각각에 여러 함수들을 순서적으로 적용하려고 할 때
[(h . g . f) x | x <- list] 혹은 [h(g(f(x))) | x <- list]
- 위의 오퍼레이션을 수행 한 후 list는 원래의 값을 그대로 유지함 : immutable

- Haskell의 lazy evaluation

take 3 [1..] = 1:take 3 [2..] = 1:2:take 1 [3..] = 1:2:3:take 0 [4..] = 1:2:3:[]
= [1.2.3.4]

- Take 함수의 두 번째 인수가 무한 개의 원소로 구성되는 데이터인 경우
- 무한 개 혹은 매우 큰 리스트의 데이터를 필요한 만큼 앞 부분에서 처리함
- 즉, 리스트 전체를 다루는 것이 아니라 필요한 갯 수만을 처리함

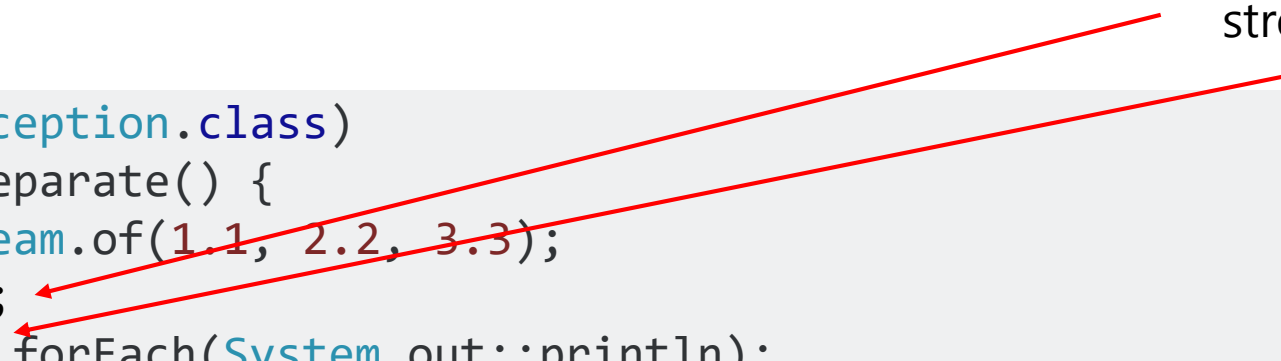
Java stream과 Haskell List의 비교

- Haskell의 List Comprehension 과 유사성
 - Lazy evaluation
 - 자체적으로 Iteration (generation) 기능을 내포함
 - No index. 오직 iteration에 의해 차례대로 원소를 제공함
 - 리스트의 모든 데이터를 전부 처리할 필요가 없으며, 필요한 만큼의 데이터를 처리함
 - Haskell의 take
 - Stream의 limit
 - 데이터의 처리를 위해 여러 함수들이 적용될 수 있음
 - 여러 함수의 합성 : 합성된 순서대로 수행됨
 - Pipeline 형태의 프로세싱
 - 대표적인 함수로서 map, filter, reduce, take(limit) 등의 함수가 적용됨
 - Immutable data structure
 - Stream 내의 원소들은 변화하지 않으며, 스트림 처리 후에도 스트림을 원 상태를 유지함
 - 필요에 따라 한 스트림으로부터 복사 및 수정하여 새로운 스트림을 생성할 수 있음
- Haskell List와의 차이점
 - Stream의 특징 "each stream is consumed only once"
 - 재귀적 형태의 프로그래밍을 하는 경우가 적음

Stream consumed once

Two
different
streams

```
@Test(expected=IllegalStateException.class)
public void multipleFilters_separate() {
    Stream<Double> ints = Stream.of(1.1, 2.2, 3.3);
    ints.filter(d -> d > 1.3);
    ints.filter(d -> d > 2.3).forEach(System.out::println);
}
```



```
@Test
public void multipleFilters_piped() {
    Stream<Double> ints = Stream.of(1.1, 2.2, 3.3);
    ints.filter(d -> d > 1.3) .filter(d -> d > 2.3) .forEach(System.out::println);
}
```

```
public void multipleFilters_separate() {
    Stream<Double> ints = Stream.of(1.1, 2.2, 3.3);
    ints = ints.filter(d -> d > 1.3);
    ints.filter(d -> d > 2.3).forEach(System.out::println);
}
```


Java Stream의 Single-Usage

A stream should be **operated on (invoking an intermediate or terminal stream operation) only once**. This rules out, for example, "forked" streams, where the same source feeds two or more pipelines, or multiple traversals of the same stream. **A stream implementation may throw `IllegalStateException` if it detects that the stream is being reused**. However, since some stream operations may return their receiver rather than a new stream object, it may not be possible to detect reuse in all cases.